一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
VLD2020-64,CPSY2020-47,RECONF2020-83(2021-01)

データ生成プログラムを利用したデータ構造の推定に基づく変異ベースファジング

難波　学之†　　石浦菜岐佐†

† 関西学院大学 理工学部　〒669–1337 兵庫県三田市学園 2–1

あらまし　本稿では, ソフトウェアのセキュリティを対象とした変異ベースファジングの効率を向上させる一手法として, シードとなるデータを生成するプログラムを利用してそのデータ構造を推定する方法を提案する. 変異ベースファジングは汎用性の高いテスト手法であるが, 有効なテストデータを生成するためには, テストデータの文法構造やテスト対象のプログラムの情報を利用する必要がある. 本手法では, バイナリデータを対象に, テストデータを生成できるプログラムを一つ選択し, そのプログラムの動作をデバッガで監視してデータを出力する関数の動的コールグラフを構築する. このコールグラフはそのデータの文法構造を表していると考えられるので, これを利用してデータの「チャンク」単位での削除, 交換, 複写を変異として行う. 本手法を Ruby で実装し, ImageMagick 6.9.7-4 の png から bmp への変換と ffmpeg 3.2.3 の wav から mp4 への変換に対して実験を行なった結果, ランダムな変異を行う場合に比べて高い頻度でターゲットプログラムのエラーを引き起こすことができた.

キーワード　ファジング, 変異ベース, データ生成プログラム

Mutation-Based Fuzzing Using Data Structure Captured via Data Generator

Noriyuki NAMBA† and Nagisa ISHIURA†

† Kwansei Gakuin University, Gakuen 2–1, Sanda, Hyogo, 669–1337, Japan

Abstract　This paper presents a method to enhance the performance of mutation-based fuzzing by inferring the data structure of seed data from the behavior of data generator programs. Although mutation-based fuzzing is a versatile test method, it must be armed with feedback from the target program or information regarding the format of the test data. In the proposed method, an appropriate program that can generate data of the format under test is run under the control of a debugger to construct a dynamic call graph regarding functions for data output, which is regarded to represent a grammatical structure of the data. By using the information, mutation such as deletion, exchange, duplication of the data at coarser grain level become possible. A fuzzing tool has been implemented in Ruby, which triggered more errors than a random mutation tool on png to bmp conversion of ImageMagick 6.9.7-4 and wav to mp4 conversion of ffmpeg 3.2.3.

Key words　Fuzzing, Mutation Fuzzing, Data Generator

1. はじめに

ソフトウェアの脆弱性は, 社会に深刻な影響をもたらすものとして近年大きな問題となっている. このため, ソフトウェアはその機能面だけでなく, 脆弱性の観点からも徹底的にテストすることが必須となっているが, 開発者が想定しないような入力による攻撃をテストケースとして人手で準備することは容易ではない.

ファジング (fuzzing) は, 自動的に大量のテストデータを生成して対象プログラムををテストする手法であり, 脆弱性検出の有効な一手法である [1]. ファジングには, 仕様書等に基づいてテストデータを一から生成する生成ベース手法と, 既存の入力データ (以下ではシードデータと呼ぶ) の一部を変異させることによってテストデータを生成する変異ベース手法が存在する.

変異ベース手法はどのような対象プログラムにも適用できるという点で汎用性が高いが, 生成ベース手法に比べてテストデータの通過率が低い (不正なデータとしてエラー処理で棄却されてしまう確率が高い) ことや, テストデータの構造を意識していないためにテストの網羅率が低いことが課題となる. これを解決するため, 対象プログラムのテストカバレッジを変異にフィードバックする手法 [3] [4] や, 変異処理系にデータの構造に関する情報を与える手法 [5] [6] が提案されている.

これに対し文献 [2] では, テストデータを生成するプログラム (対象プログラムが JPEG ファイルを入力するのであれば, JPEG

ファイルを生成する何らかのプログラム) を利用して変異ベースファジングの効率向上を図るアプローチを提案している. この手法では, データ生成プログラム中の低レベルデータ出力関数をデバッガで監視することにより, シードデータ中のデータ項目のバイト数を推定し, それを利用した変異を行っている.

本稿では, 文献 [2] の手法を拡張してテストデータの文法構造を推定し, より大きなデータのかたまりを認識して変異を行う手法を提案する.

2. ファジング

2.1 脆弱性検出のためのファジング

ファジングは, 自動生成した大量の入力データによってプログラムのテストを行う手法である. ソフトウェアの品質検査全般に利用できるが, 特に脆弱性を検出する有効な一手法である. 脆弱性の検出を目的としたテストの場合には, 自動生成したデータをターゲットとなるプログラムに入力し, プログラムのクラッシュ, 無限ループ, メモリ異常, シグナルの発生等を監視する.

プログラムへの入力は, ファイルやネットワークを介したものから, コマンドライン引数, シェル変数等多岐に渡る. 入力データの形式についても HTML や CSS 等のテキストデータから JPEG や MP4 等のバイナリデータまであらゆるものが可能である.

2.2 ファジングのアプローチ

ファジングのためのデータの生成方法には生成ベース手法と変異ベース手法がある [1]. 生成ベース手法は, データの仕様や文法に基づいてプログラムにより一からデータを生成するものである. この手法は, 正当なデータや正当なデータとわずかに異なる不正なデータを生成できるため, 網羅率の高い検査が行えるという利点がある. 一方で, 生成ツールの開発コストが高い上, データ形式毎に生成ツールを作成しなければならない. 変異ベース手法は, 既存のデータの一部を書き換える (変異させる) ことによってテスト用データを生成する手法である. この手法は実装が容易であり, 変異処理系を作成すればそれを種々のデータ形式にも適用できるという意味で汎用性が高い. しかし, データ形式に準拠しないデータが生成され易いため, データがターゲットプログラムのエラー処理で棄却されたり, 網羅率の高いテストが行えないという欠点がある.

この欠点を解決して変異ベースファジングの効率を向上させる手法が多く提案されている. 一つ目のアプローチは, 図 1 (a) のように, 生成したデータを入力した時のターゲットプログラムの動作情報をフィードバックする方法である. 文献 [3] [4] では, 生成したデータによるターゲットプログラムのコード網羅率を計測し, それが上昇するようにデータの変異を制御している. 二つ目のアプローチは, 図 1 (b) のように, 変異に入力データのフォーマット情報を用いる方法である. 文献 [6] では, 入力データ中の意味のあるデータのかたまり「チャンク」を特定するための情報を, 文献 [5] では入力データの文法を与えることにより, テストに有効な変異を行う手法を提案している. しかし, フォーマット情報を与えて変異を行うことは, ある意味生成ベース手法と等価であり, 同手法と同じ課題を持つことになる.

2.3 データ生成プログラムを利用するアプローチ

文献 [2] は新たな手法として, 図 1 (c) のように, シードデータを生成する何らかのプログラムの動作を監視することによってデータのフォーマット情報を推定し, それをデータの変異に利用する手法を提案している. この手法の詳細を図 2 に示す. まずシードデータを生成するプログラムをデバッガで監視し, データを出力する関数 (C 言語の fwrite や fput 等の関数; 図中 w) が呼び出される毎にその引数を取得する. 引数に指定されている出力データのバイト数やデータ数からデータ項目の境界やデータの繰り返しの有無を推定するとともに, データそのものからデータの型 (文字データか数値データか) を推定し, それに基づいた変異を行う.

この手法は, テキストデータのように区切り記号からデータ項目の位置を推定できないバイナリデータに対して有効である. しかし, この手法が利用しているのは最下層の出力関数が出力するデータ項目に関する情報だけであり, データの文法構造やデータのチャンクに関する情報は抽出できない.

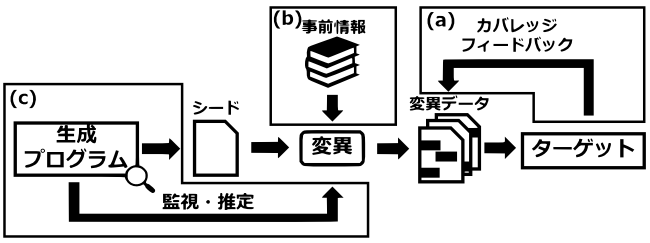


図 1　変異ファジングのアプローチ

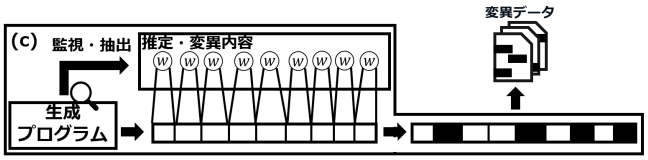


図 2　文献 [2] の手法

3. 生成プログラムを利用したデータ構造の推定に基づく変異

3.1 概　　要

本稿では, 文献 [2] の手法を拡張し, シードデータ中のデータ項目の情報だけでなく, その文法構造を推定して, その情報に基づいたデータの変異を行う手法を提案する.

提案手法の処理の流れを図 3 に示す. 文献 [2] と同様, デバッガによりデータ生成プログラム中の出力関数の呼び出しを監視するが, 本手法では呼び出し時点のスタックのダンプからその関数の呼び出し元を再帰的に求める. この情報から動的なコールグラフを構築すると, それは出力されるデータの文法構造を表していると考えられるので, ここからデータのチャンクの境界が推定できる. この情報を利用して, データのチャンク単位で

の交換, 削除, 複写を行うことにより, ファジングの効率向上を図る.

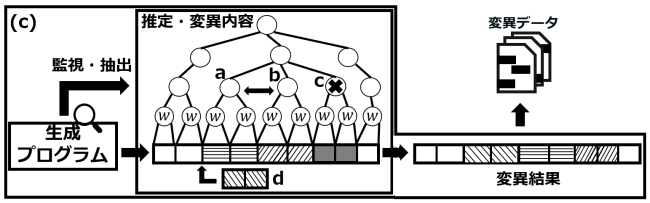


図 3　提案手法の処理の流れ

3.2　関数の動的コールグラフの構築とそれに基づくデータ構造の推定

データ生成プログラムにおいて, データの出力を行なっている関数の動的コールグラフと出力されるデータの関係を図 4 に示す. 関数 m は関数 d, e, f, e をこの順に呼び出している. ここで γ^j は関数 γ の j 回目の呼び出しを表す. 最下層の関数 w は C 言語の fwrite や fput のようなデータ出力関数であり, $w^1, w^2, w^3, ...$ はそれぞれデータ項目 $d_1, d_2, d_3, ...$ を出力している. a^1 は w^1 と w^2 の呼び出しによりデータ $D1 = d_1 d_2$ を出力しており, d^1 は a^1 と w^3 の呼び出しによりデータ $D_2 = D_1 d_3 = d_1 d_2 d_3$ を出力している. このように, データ出力関数の動的コールグラフは出力データの文法構造を表す導出木になっていると見ることができる.

一つの関数が一つの意味のあるデータの塊, すなわちチャンクを出力するので, これを利用すればバイナリデータ中のどこからどこまでが一つのチャンクかを推定できる. また, そのチャンクが同じ関数から出力されていれば, そのチャンクは何らかの意味で同じ役割を持っているものと推測することができる.

この動的コールグラフはデバッガを使って抽出したスタックのトレースから構築できる. 動的コールグラフの作成の流れを図 5 に示す. まず, デバッガでデータ出力関数 w を監視し, w の呼び出しがあった時点のスタックダンプを取得する. そこから関数呼び出しの戻り番地を再帰的に取得すれば, 関数の呼び出し系列を取得できる (図 5 (a)). 全ての w の呼び出しについて関数の呼び出し系列を列挙するが, この際 w の呼び出しには出現順に $w^1, w^2, ...,$ と呼び出し番号を付与する. このリストより, 関数呼び出しを節点とし, 呼び出し元から呼び出し先に有向枝を設けたグラフを作成する. 全ての呼び出しの系列において最初に呼び出される関数は同じ (図 5 (b) 中の m) であるとすると, このグラフは m を根とする木となる. この木において, 親が同じで関数が同じ節点があれば図 5 (c) のように縮約を行う. これを繰り返すことにより, 図 5 (d) のような動的コールグラフを構築できる.

3.3　データ構造の推定を使用した新たな変異

前章の方法で取得した情報を元に新たな変異を導入する. 本稿で追加する変異は, 推定したチャンク単位での (a) 交換, (b) 削除, (c) 複写である.

(a) 交換

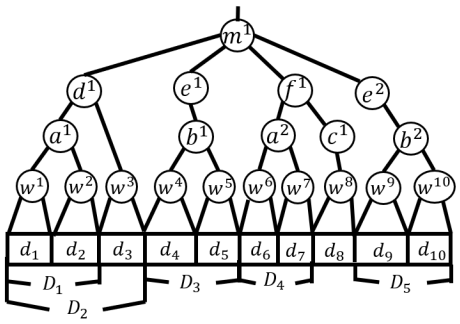
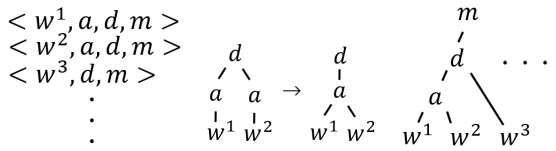


図 4　動的コールグラフと出力データ

w: < wの戻り番地, aの戻り番地, dの戻り番地, mの戻り番地 >
(a 内)　(d 内)　(m内)
↓
$< w, a, d, m >$

(a) バックトレースの結果



(b) トレース結果の一覧　(c) ツリーの縮約　(d) 最終結果

図 5　動的コールグラフの構築

動的コールグラフにおいて関数が同じ節点を 2 つランダムに選び, その位置を交換する (図 6 (a) の a と b). これにより, 2 つの節点から生成されるチャンクの位置が交換される.

(b) 削除

動的コールグラフの節点を 1 つランダムに選び, その位置の節点を削除する (図 6 (b) の a). これにより, 節点から生成されるチャンクを削除する. ただし, データに不可欠なチャンクを削除するとエラーで棄却される確率が高くなるため, 同じ関数の存在が複数確認されている節点にのみ対象を絞る.

(c) 複写

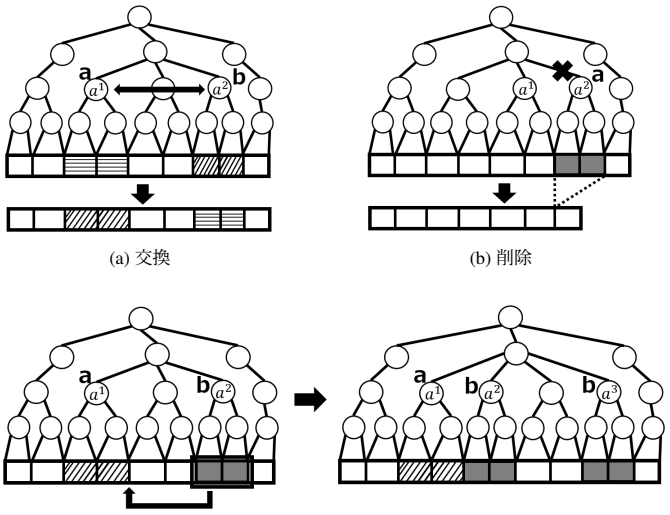
動的コールグラフにおいて関数が同じ節点を 2 つランダムに選び, 片方の節点の複製をもう片方の節点の直後に追加する (図 6 (c) の a(追加先) と b(複写内容)).

4.　実装と実験結果

本手法に基づくテストシステムを, Ruby2.4.5 で実装した. 動的コールグラフの抽出を行うためのデバッガには GDB を使用した. 本手法の評価を行うために, ランダムな変異手法, および文献 [2] のデータ項目の情報を利用した変異手法との比較実験を行なった. 実験の流れを図 7 に示す. ランダムな変異手法では, 生成プログラムが出力したデータをシードとして, その中のバイトをランダムに選択して変異させた. 文献 [2] の手法と本手法では, 生成プログラムをデバッガで監視して得られる情報からどのシードデータ中のどの部分をどう変異させるかという「変異情報」を一旦出力し, これを入力として生成プログラムを再度実行して変異したデータを生成した.

テスト対象としたターゲットプログラムは

- ImageMagick 6.9.7-4 の png から bmp への変換



(a) 交換　(b) 削除

(c) 複写

図 6　提案手法の変異

- ffmpeg 3.2.3 の wav から mp4 への変換

の 2 つである. png ファイルと mp4 ファイルの生成プログラムとしては,

- ImageMagick 6.9.7-4 の png から bmp への変換
- opusdec 1.1.2 の opus から wav への変換

を用いた.

ランダムな手法では, 100 バイトに 1 バイトの割合で値を 0 255 のランダムな値に変更した. 文献 [2] の手法では, まず, 抽出したデータ出力関数の中で各関数のデータ出力量に応じて 1/10 もしくは 1.2/10 の確率で変異する関数を選択する. そして, 選ばれた出力関数で出力されているデータ群の中の 1 つを変異した. 変異内容は変異前の数値の ±10, 特殊文字, 最大最小値付近の数値, ビット反転, 乱数の中でランダムに実行した. 本手法では, 変異の回数 (交換, 削除, 複写を行う箇所の数) の上限を ImageMagick では 3 回, opusdec では 15 回とした.

ImageMagick に対する実験結果を表 1 に示す. #test はターゲットに入力した変異データの数, #pass はそのうち変異データがターゲットプログラムの入力処理を通過した (入力データに対するエラー処理で棄却されなかった) ものの数, #error はターゲットプログラムのエラーを引き起こしたものの数であり, うち abort が異常終了の件数, time-out が無限ループによるタイムアウトの件数である. time (s) はテストデータの生成とテストの実行に要した時間の合計である. この実験で検出された abort のエラーは全て assertion failure によるものあった. (a)(b)(c) 全ての変異で, データの通過数, エラー発生数ともランダムな変異手法と文献 [2] の手法を上回った. 実行時間は従来法に比べて増加しているが, これは入力データがエラー処理で棄却されずに本体の処理が行われたデータの数が多いことが主因と考えられる.

ffmpeg に対する実験結果を表 2 に示す. 従来法に比べ (a) では #pass と#error を増加させることができた. また (c) では, 従来法では検出できなかったプログラムの異常終了 (assertion failure によるもの) を 1 件検出できた. テストの実行に要する時間に関しては, ImageMagick の場合と同様の傾向が見られる.

表 1　実験結果 (ImageMagick png → bmp)

	#test	#pass	#error		time (s)
			abort	time-out	
ランダム	2000	399	141	0	1645
文献 [2]	2000	303	127	0	1641
本手法 (a) 構造交換	2000	719	565	0	1729
本手法 (b) 構造削除	2000	929	929	0	2018
本手法 (c) 構造複写	2000	1554	1527	0	1618

表 2　実験結果 (ffmpeg wav → mp4)

	#test	#pass	#error		time (s)
			abort	time-out	
ランダム	2000	1624	0	7	3470
文献 [2]	2000	1681	0	18	3511
本手法 (a) 構造交換	2000	1876	0	29	8333
本手法 (b) 構造削除	2000	1434	0	0	3411
本手法 (c) 構造複写	2000	1557	1	0	5476

シードデータに関するより上位の構造を推定して変異に反映することにより, ファジングの性能を向上させることができると考えられる.

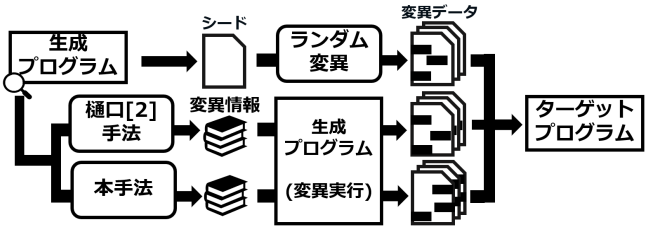


図 7　比較実験の内容

5. む　す　び

本稿では, シードデータを生成するプログラムを利用してデータの文法構造を推定し, その情報に基づいたデータの変異を行う手法を提案した. 評価実験の結果, 本手法により生成したテストデータは, ターゲットプログラムのエラー処理を通過する率やターゲットプログラムにエラーを引き起こす率が従来法よりも高かった.

本手法をさらに拡張することにより, シードデータ中に同じ種類のチャンクが繰り返し出現している等のさらに高位のデータ構造を推定することができると考えられる. また, 単一のシードデータではなく, 複数のシードデータを用いることによってもデータ構造に関するより高位の情報を推定できると考えられる. このような情報を利用してファジングの効率を向上させることが今後の課題として挙げられる.

謝辞　本研究を行うにあたり, システムの基盤の作成に多大なる御協力を頂いた樋口 瑛子氏, 本研究の実験等の様々な御助力を頂いた伊勢 斗馬氏, その他御助言御助言や御協力を頂いた関西学院大学理工学部石浦研究室の諸氏に感謝致します.

文　　献

[1] Michael Sutton, Adam Greene, Pedram Amini 著, ドキュメントシステム＋伊藤裕之 訳 : ファジング ブルートフォースによる脆弱

性発見手法, 毎日コミュニケーションズ (June 2008).
[2] 樋口瑛子, 石浦菜岐佐, 難波学之: “データ生成プログラムを利用したデータ項目の型推定に基づく変異ベースファジング,” 電子情報通信学会技術研究報告, VLD2020-87 (Jan. 2020).
[3] Dong Fangquan, Dong Chaoqun, Zhang Yao, Lin Teng: “Binary-oriented hybrid fuzz testing” in *IEEE International Conference on Software Engineering and Service Science (ICSESS 2015)*, pp. 345–348 (Sept. 2015).
[4] American Fuzzy Lop (online), `http://lcamtuf.coredump.cx/afl/` (accessed 2020-11-11).
[5] Junjie Wang, Bihuan Chen, Lei Wei and Yang Liu: “Superion: Grammar-Aware Greybox Fuzzing,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE 2019)*, pp.724–735 (May. 2019).
[6] Van-Thuan Pham, Marcel Bohme, Andrew E. Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury : “Smart Greybox Fuzzing,” in *IEEE Transactions on Software Engineering (Early Access)*, pp. 1–17 (Sept. 2019).